Chapter 11
Genetic algorithm and programming

From the relevant discussions on function optimization covered in Chapters 3, 5, and 10, we by now should have realized that to find the global minimum or maximum of a multivariable function is in general a formidable task even though a search for an extreme of the same function under certain circumstances is achievable. This is the driving force behind the never-ending quest for newer and better schemes in the hope of finding a method that will ultimately lead to the discovery of the shortest path for a system to reach its overall optimal configuration.

The *genetic algorithm* is one of the schemes obtained from these vast efforts. The method mimics the evolution process in biology with inheritance and mutation from the parents built into the new generation as the key elements. Fitness is used as a test for maintaining a particular genetic makeup of a chromosome. The scheme was pioneered by Holland (1975) and enhanced and publicized by Goldberg (1989). Since then the scheme has been applied to many problems that involve different types of optimization processes (Bäck, Fogel, and Michalewicz, 2003). Because of its strength and potential applications in many optimization problems, we introduce the scheme and highlight some of its basic elements with a concrete example in this chapter. Several variations of the genetic algorithm have emerged in the last decade under the collective name of evolutionary algorithms and the scope of the applications has also been expanded into multiobjective optimization (Deb, 2001; Coello Coello, van Veldhuizen, and Lamont, 2002). The main purpose here is to introduce the practical aspects of the method. Readers interested in its mathematical foundations can find the relevant material in Mitchell (1996) and Vose (1999).

We can even take one further step. Instead of encoding the possible configurations into chromosomes, possible computing operations can be selected or altered, resulting in the program encoding itself. This direct manipulation or creation of optimal programs based on the evolution concept is called *genetic programming*, which was conceptualized 40 years ago (Fogel, 1962) and matured more recently (Koza, 1992). We will not be able cover all the details of genetic

programming here but will merely highlight the concept toward to the end of the chapter.

11.1 Basic elements of a genetic algorithm

The basic idea behind a genetic algorithm is to follow the biological process of evolution in selecting the path to reach an optimal configuration of a given complex system. For example, for an interacting many-body system, the equilibrium is reached by moving the system to the configuration that is at the global minimum on its potential energy surface. This is single-objective optimization, which can be described mathematically as searching for the global minimum of a multivariable function $g(r_1, r_2, \ldots, r_n)$. Multiobjective optimization involves more than one equation, for example, a search for the minima of $g_k(r_1, r_2, \ldots, r_n)$ with $k = 1, 2, \ldots, l$. Both types of optimization can involve some constraints. We limit ourselves to single-objective optimization here. For a detailed discussion on multi-objective optimization using the genetic algorithm, see Deb (2001).

In this section, we describe only a binary version of the genetic algorithm that closely follows the evolutionary processes. The advantage of the binary genetic algorithm lies in its simplicity, and it articulates the evolution in the forms of binary chromosomes. Later in the chapter, we will also introduce the algorithm that uses real numbers in constructing a chromosome. The advantage of the real genetic algorithm is that the search is done with continuous variables, which better reflects the nature of a typical optimization problem.

In the binary algorithm, each configuration of variables $(r_1, r_2, \ldots, r_n)$ is represented by a binary array, This array can be stored on a computer as an integer array with each element containing a decimal number 1 or 0, or as a boolean array with each element containing a bit that is set to be true (1) or false (0). We will use boolean numbers in actual computer codes but use decimal 0s and 1s when writing equations for convenience.

Several steps are involved in a genetic algorithm. First we need to create an initial population of configurations, which is called the initial gene pool. Then we need to select some members to be the parents for reproduction. The way to mix the genes of the two parents is called crossover, which reflects how the genetic attributes are passed on. In order to produce true offspring, each of the parent chromosomes is cut into segments that are exchanged and joined together to form the new chromosomes of the offspring. After that we allow a certain percentage of bits in the chromosomes to mutate. In the whole process, we use the fitness of each configuration based on the cost (the function to be optimized) $g(r_1, r_2, \ldots, r_n)$ as the criterion for selecting parents and sorting the chromosomes for the next generation of the gene pool. In each of the three main operations (selection, crossover, and mutation) in each generation, we make sure that the elite configurations with the lowest costs always survive.

Creating a gene pool

The initial population of the gene pool is typically created randomly. A sorting scheme is used to rank each of the chromosomes according to its fitness. The following method shows an example of how to create the initial population of the gene pool.

```
// Method to initialize the simulation by creating the
// zeroth generation of the gene population.

  public static void initiate(){
    Random rnd = new Random();
    boolean d[][] = new boolean[ni][nd];
    boolean w[] = new boolean[nd];
    double r[] = new double[nv];
    double e[] = new double[ni];
    int index[] = new int[ni];

    for (int i=0; i<ni; ++i) {
      for (int j=0; j<nv; ++j) r[j] = rnd.nextDouble();
      e[i] = cost(r);
      index[i] = i;
      w = encode(r,nb);
      for (int j=0; j<nd; ++j) d[i][j] = w[j];
    }
    sort(e,index);
    for (int i=0; i<ng; ++i){
      f[i] = e[i];
      for (int j=0; j<nd; ++j) c[i][j] = d[index[i]][j];
    }
  }
```

There are several issues that need to be addressed during the initialization of the gene pool. The first is the size of the population. Even though each individual configuration in the gene pool is represented by a distinct chromosome, a good choice of the population size optimizes the convergence of the simulation. If the population is too small, it will require more time to sample the entire possible configuration space; but if the population is too large, it takes more time to create a new generation each time. The second issue concerns the quality of the initial gene pool. If we just used all the configurations created randomly, the quality of the initial gene pool would be low and that means a longer convergence time. Instead, as shown in the above example, we usually create more chromosomes initially in order to give us a choice. For example, if we want to have a population of n_g chromosomes in the gene pool, we can randomly generated a larger number of chromosomes and then select the best n_g chromosomes that have the lowest costs. A typical choice is to have $n_i = 2n_g$ chromosomes from which to choose. We have to encode each configuration into a chromosome and also evaluate its corresponding cost. These two issues will be discussed later in the section. After we obtain the costs of all the configurations, we can rank them accordingly through a sorting scheme. Because we want the ranking recorded and used to relabel the

chromosomes, we need to assign a ranking index in the sorting process. The following method shows how to achieve such a sorting.

```
// Method to sort an array x[i] from the lowest to the
// highest with the original order stored in index[i].

  public static void sort(double x[], int index[]){
    int m = x.length;
    for (int i = 0; i<m; ++i) {
      for (int j = i+1; j<m; ++j) {
        if (x[i] > x[j]) {
          double xtmp = x[i];
          x[i] = x[j];
          x[j] = xtmp;
          int itmp = index[i];
          index[i] = index[j];
          index[j] = itmp;
        }
      }
    }
  }
```

Note that the index used here in the input is in a natural order and that in the output is in increasing order of the associated costs of all the configurations. This sorting scheme is also used in rearranging the chromosomes in other places when needed.

Encoding a configuration

Because we want to model the genetic process accurately, we need to convert each configuration $(r_1, r_2, \ldots, r_n)$ into a chromosome through an encoding process. We will assume that the variables r_i are in the region [0, 1], that is, $0 \leq r_i \leq 1$ for $i = 1, 2, \cdots, n$. This does not affect the generality of the discussion as long as the variables are bound in a finite region so they can always be cast back into the region [0, 1] by a linear transformation.

We can represent any variable $r_i \in [0, 1]$ by a binary string in which each bit is set to a true or false value. If the kth bit is true, there is a fraction $1/2^k$ contributed to the variable, which can then be expressed as

$$r_i = \frac{y_{i1}}{2} + \frac{y_{i2}}{4} + \frac{y_{i3}}{8} + \cdots = \sum_{j=1}^{\infty} \frac{y_{ij}}{2^j}, \tag{11.1}$$

where y_{ij} is a decimal integer equal to either 0 or 1. We can truncate the binary string at a selected number m, whose value depends on how accurate we want r_i to be. Then we have

$$r_i \simeq \sum_{j=1}^{m} \frac{y_{ij}}{2^j}. \tag{11.2}$$

For example, if $r_i = 0.93$, We have $y_{i1} = y_{i2} = y_{i3} = 1$ and $y_{i4} = 0$, and if $r_i = 0.6347$, we have $y_{i1} = y_{i3} = 1$ and $y_{i2} = y_{i4} = 0$, for $m = 4$. The maximum error

created in r_i is $\pm 1/2^{m+1}$. The process of generating y_{ij} is called *encoding*, which is accomplished with

$$y_{ij} = \text{int}\left[2^{j-1} r_i - \sum_{k=1}^{j-1} \left(2^{j-k-1} y_{ik}\right)\right], \tag{11.3}$$

for $j = 2, 3, \ldots, m$, where the operation int rounds the value inside the square bracket to the nearest decimal integer (either 0 or 1), with $i = 1, 2, \ldots, n$. Note that $y_{i1} = \text{int}[r_i]$.

The encoding is a process of representing r_i by a binary array with each element containing a bit that is set to be true or false, corresponding to a decimal integer 1 or 0. We call this binary array y_{ij} for $j = 1, 2, \ldots, m$ the ith *gene* of the *chromosome* that is a binary representation of the entire real array $(r_1, r_2, \ldots, r_n)$. The following method encodes the array r_i for $i = 1, 2, \ldots, n$ into a binary chromosome $w = \{y_{11} \ldots y_{1m}\, y_{21} \ldots y_{2m}\, y_{n1} \ldots y_{nm}\}$.

```
// Method to encode an array of n real numbers r[i] in
// [0,1] into an n*m binary representation w[j].
  public static boolean[] encode(double r[], int m) {
    int n = r.length;
    boolean w[] = new boolean[n*m];
    for (int i = 0; i<n; ++i) {
      double sum = r[i];
      w[i*m] = false;
      if((int)(0.5+sum) == 1) w[i*m] = true;
      double d = 2;
      for (int j = 1; j<m; ++j) {
        if(w[i*m+j-1]) sum -= 1/d;
        w[i*m+j] = false;
        if((int)(0.5+d*sum) == 1) w[i*m+j] = true;
        d *= 2;
      }
    }
    return w;
  }
```

Of course, the reverse of the encoding process is also necessary when we need to use the configuration information in the evaluation of the cost function or output the final configurations. To decode a chromosome, we can use Eq. (11.2). For example, if we have a chromosome $w = \{1010111001\ldots\}$ for $m = 10$, the corresponding variable

$$r_1 \simeq \frac{1}{2} + \frac{1}{2^3} + \frac{1}{2^5} + \frac{1}{2^6} + \frac{1}{2^7} + \frac{1}{2^{10}} \simeq 0.6807. \tag{11.4}$$

Note that the uncertainty in r_1 here is determined by the choice of m, given by $\pm 1/2^{m+1} \simeq \pm 0.0005$ in this case. This decoding process can be achieved quite easily in a program. The following is an example for decoding a binary array.

```
// Method to decode an array of n*m binary numbers w[j]
// into an array of n real numbers r[i].

  public static double[] decode(boolean w[], int m) {
    int n = w.length/m;
    double r[] = new double[n];
    for (int i = 0; i<n; ++i) {
      double d = 2;
      double sum = 0;
      for (int j = 0; j<m; ++j) {
        if(w[i*m+j]) sum += 1/d;
        d *= 2;
      }
      r[i] = sum + 1/d;
    }
    return r;
  }
```

Now we know how to create a chromosome w for a real array $(r_1, r_2, \ldots, r_n)$. We can then use the encoding scheme to create an initial population that is sorted with the cost function. After we have all these tools, we can accomplish the three main operations, selection, crossover, and mutation, in a genetic algorithm.

Selection operation

We need to able to select a fraction of the chromosomes from the given gene pool to pass on their genes. This models the natural reproduction process. If we take Darwin's concept literally, the chromosomes with better costs are the ones most likely to survive. Several methods have been designed to follow this concept in selecting the parents. A simple choice is to use the best half of the chromosomes from the entire population according to their costs. Then we can randomly select one pair after another from this parent pool to create a certain number of offspring. Another choice is to select a chromosome from the entire population with a probability based on a weight assigned according to either its ranking in cost or its relative cost. For a comparative study of these choices, see Goldberg and Deb (1991).

The most popular method for creating the parents is to hold tournaments, in each of which the two or more participants are selected at random with the winner of each tournament being the participant with the best cost. The following method illustrates how to select the winners from one-on-one matches.

```
// Method to run tournaments for selecting the parents.

  public static void select() {
    int index[] = new int[ng];
    boolean d[][] = new boolean[ng][nd];
    double e[] = new double[ng];

    for (int i=0; i<ng; ++i){
      for (int l=0; l<nd; ++l) d[i][l] = c[i][l];
      e[i] = f[i];
```

```
      index[i] = i;
    }
    shuffle(index);
    int k = 0;
    for (int i=0; i<nr; ++i) {
      if (e[index[k]] < e[index[k+1]]){
        for (int l=0; l<nd; ++l) c[i][l]=d[index[k]][l];
        f[i] = e[index[k]];
      }
      else {
        for (int l=0; l<nd; ++l) c[i][l]=d[index[k+1]][l];
        f[i] = e[index[k+1]];
      }
      k += 2;
    }
  }
```

Note that we have allowed each member in the pool to participate, but in only one match. Half of the chromosomes in the pool ($n_r = n_g/2$) are selected to be the parents without any duplication. The above method will always result in a copy of the best chromosome being and retained the worst being eliminated. Some other tournament schemes do allow duplications, for example, to have two copies of the best chromosome in the parent pool by letting each chromosome participate in two matches. Shuffling is used to mix up the indices before the matches so that the participants are randomly drawn for a match with an equal probability. Here is how we shuffle the indices.

```
// Method to shuffle the index array.
  public static void shuffle(int index[]){
    int k = index.length;
    Random rnd = new Random();
    for (int i = 0; i<k; ++i) {
      int j = (int)(k*rnd.nextDouble());
      if (j!=i) {
        int itmp = index[i];
        index[i] = index[j];
        index[j] = itmp;
      }
    }
  }
```

After we have completed the selection, we are ready to make some new chromosomes.

Crossover operation

So far we have found ways of creating a gene pool based on randomly assigned bits and then selecting certain genes to be the parents. The next step is to devise a way of exploring the cost surface. There are two operations in the genetic algorithm that effectively look over the entire variable space. The first operation is called

crossover, which is achieved by mimicking the reproduction processes in Nature. We can choose a pair of parents from the parent pool, cut each chromosome of the parents into two segments at a selected point, and then join the right segment of one parent to the left segment of the other, and vice versa, to form two new chromosomes for the offspring. For example, if the chromosomes of two parents are $w_1 = \{01101010\}$ and $w_2 = \{10101101\}$, the corresponding chromosomes of the offspring are $w_3 = \{01101101\}$ and $w_4 = \{10101010\}$, respectively, with the crossover point taken as the middle of the chromosomes. Crossover is one the most effective ways of exploring the cost surface of all the possible configurations. What we have described here is the single-point crossover scheme. There are other types of crossover schemes and the readers can find them in the related literature (Spears, 1998). The following method is an implementation of the single-point crossover.

```
// Method to perform single-point crossover operations.
  public static void cross() {
    Random rnd = new Random();
    int k = 0;
    for (int i=nr; i<nr+nr/2; ++i) {
      int nx = 1 + (int)(nd*rnd.nextDouble());
      for (int l=0; l<nx; ++l){
        c[i][l] = c[k][l];
        c[i+nr/2][l] = c[k+1][l];
      }
      for (int l=nx; l<nd; ++l){
        c[i][l] = c[k+1][l];
        c[i+nr/2][l] = c[k][l];
      }
      k += 2;
    }
  }
```

Note that each time we have taken two members from the parent pool in order to create two offspring, who are made from single-point crossover operations. We can also choose the parents differently from the pool (Goldberg, 1989). After we complete the reproduction of $n_r = n_g/2$ offspring, we need to rearrange all the chromosomes, parents and offspring, into an increasing order of the cost. This is necessary in preparing the gene pool before any mutation takes place. The following method shows how to rearrange the chromosomes.

```
// Method to rank chromosomes in the population.
  public static void rank() {
    boolean d[][] = new boolean[ng][nd];
    boolean w[] = new boolean[nd];
    double r[] = new double[nv];
    double e[] = new double[ng];
    int index[] = new int[ng];
```

```
    for (int i=0; i<ng; ++i) {
      for (int j=0; j<nd; ++j){
        w[j] = c[i][j];
        d[i][j] = w[j];
      }
      r = decode(w,nb);
      e[i] = cost(r);
      index[i] = i;
    }
    sort(e,index);
    for (int i=0; i<ng; ++i){
      f[i] = e[i];
      for (int j=0; j<nd; ++j) c[i][j] = d[index[i]][j];
    }
  }
```

Note specifically how the index array is used to help relabel the chromosomes according to the cost of each configuration.

Mutation operation

Another effective way of exploring the cost surface in the genetic algorithm is through the mutation process. This is achieved by randomly reversing the bits in the randomly selected chromosomes. There are two issues related to mutation.

First is the percentage of bits in the entire gene pool that are to be mutated in each generation. A typical case is to select about 1% of bits randomly for mutation, 0 (false) to 1 (true) and 1 (true) to 0 (false). The higher the percentage the bigger the fluctuation. However, if a larger fraction of bits is mutated in each generation, the cost surface can be explored faster, but it may mean that the best cost is skipped in the process. So in an actual calculation, we need to experiment with different percentages for a specific given problem (cost function) in order to find a moderate percentage that will allow us to explore the cost surface fast enough without missing the best cost configuration.

Another issue is the number of configurations that we would like to keep immune from mutation. Under any mutation scheme, the best chromosome is always kept unchanged. But we may also want a few of the next-best chromosomes to be immune from mutation. This slows down the exploration of the cost surface by mutation but increases the rate of convergence because those configurations may have already contain a large fraction of excellent genes in their chromosomes. In practice, we need to experiment in order to find the most suitable percentage of bits to be mutated for the specific problem. The following method is an example of performing mutation on a population with a given percentage of bits to be reversed.

```
// Method to mutate a percentage of bits in the selected
// chromosomes except the best one.

  public static void mutate() {
    int mmax = (int)(ng*nd*pm+1);
    Random rnd = new Random();
```

```
    double r[] = new double[nv];
    boolean w[] = new boolean[nd];

 // Mutation in the elite configurations
    for (int i=0; i<ne; ++i) {
      for (int l=0; l<nd; ++l) w[l] = c[i][l];
      int mb = (int)(nd*pm+1);
      for (int j=0; j<mb; ++j){
        int ib = (int)(nd*rnd.nextDouble());
        w[ib] = !w[ib];
      }
      r = decode(w,nb);
      double e = cost(r);
      if (e<f[i]){
        for (int l=0; l<nd; ++l) c[i][l] = w[l];
        f[i] = e;
      }
    }

 // Mutation in other configurations
    for (int i=0; i<mmax; ++i) {
       int ig = (int)((ng-ne)*rnd.nextDouble()+ne);
       int ib = (int)(nd*rnd.nextDouble());
       c[ig][ib] = !c[ig][ib];
    }

 // Rank the chromosomes in the population
    rank();
  }
```

Note that we have kept a certain number of elite configurations immune from mutation unless the attempted mutation improves their costs. All the configurations are rearranged according to their costs after each round of mutation operation. Now we are ready to see how the algorithm works in actual problems.

11.2 The Thomson problem

In electrostatics, charges will distribute to minimize the electrostatic energy under the equilibrium condition implied. This is the so-called Thomson theorem. For example, we can show that in a system of conductors, each forms an equipotential surface if the charge placed on each conductor is fixed and the total electrostatic energy of the system is minimized. However, a problem arises when the equilibrium configuration of a significant number of discrete charges is sought, for example, the stable geometry of n_c identical point charges confined on the surface of a unit sphere. This problem, known as the Thomson problem, originates from two complexities: the nonlinearity in the behavior of the system and a large number of low-lying energy levels.

We can obtain definite answers for some limited cases. For example, we can prove that the charges will cover the entire surface uniformly if $n_c \to \infty$ and that the symmetric geometries to keep all the charges far apart are the stable configurations for small n_c, such as an equilateral triangle for $n_c = 3$, a tetrahedron for $n_c = 4$, a twisted and stretched cube for $n_c = 8$, and so forth. However,

the problem becomes increasingly complicated as n_c increases. For example, when n_c reaches 200, the number of nearly degenerate low-lying energy levels is about 8000. The problem with a large n_c is as yet still considered unsolved. Here we want to demonstrate the application of the genetic algorithm to the Thomson problem to demonstrate the strength of the scheme. Furthermore, the calculations achieved with the genetic algorithm for $n_c \leq 200$ are the best results obtained so far (Morris, Deaven, and Ho, 1996).

Mathematically, the Thomson problem is to find the configuration that minimizes the electrostatic energy

$$U = \frac{q^2}{4\pi\epsilon_0} \sum_{i>j=1}^{n_c} \frac{1}{|\mathbf{r}_i - \mathbf{r}_j|}, \tag{11.5}$$

where q is the charge on each particle, ϵ_0 is the electric permittivity of free space, and $\mathbf{r}_i$ is the position vector of the ith charge. Because all the charges are confined on the surface of the unit sphere, $r_i \equiv 1$. We will take $q^2/4\pi\epsilon_0 = 1$, reflecting a special choice of units, for convenience. If we represent the Cartesian coordinates in terms of the polar and azimuthal angles, we have

$$\begin{aligned} x_i &= \sin\theta_i \cos\phi_i, \\ y_i &= \sin\theta_i \sin\phi_i, \\ z_i &= \cos\theta_i. \end{aligned}$$

Here we have taken the radius of the sphere to be unit. The following method is an implementation of the evaluation of the cost (total electrostatic energy of the system) in the Thomson problem.

```
// Method to evaluate the cost for a given variable array.

  public static double cost(double r[]) {
    double g = 0;
    double theta[] = new double[nc];
    double phi[] = new double[nc];

    for (int i=0; i<nc; ++i){
      theta[i] = Math.PI*r[i];
      phi[i] = 2*Math.PI*r[i+nc];
    }

    for (int i=0; i<nc-1; ++i){
      double ri = Math.sin(theta[i]);
      double xi = ri*Math.cos(phi[i]);
      double yi = ri*Math.sin(phi[i]);
      double zi = Math.cos(theta[i]);
      for (int j=i+1; j<nc; ++j){
        double rj = Math.sin(theta[j]);
        double dx = xi - rj*Math.cos(phi[j]);
        double dy = yi - rj*Math.sin(phi[j]);
        double dz = zi - Math.cos(theta[j]);
        g += 1/Math.sqrt(dx*dx+dy*dy+dz*dz);
      }
    }
    return g;
  }
```

Fig. 11.1 The stable structures of five and eight charges on a unit sphere.

Note that the input variables are assumed to be in the region [0, 1] and we therefore convert them to the correct regions with $\theta_i \in [0, \pi]$ and $\phi \in [0, 2\pi]$, for $i = 1, 2, \ldots, n_c$. To test the validity of each part of the computer code that we have developed for the Thomson problem, we search first for the geometric configurations of small systems. In Fig. 11.1 we show the stable configurations that we have obtained for $n_c = 5$ and $n_c = 8$. We obtain these stable configurations in just about 10 000 generations with the total energies $E_5 = 6.4747$ and $E_8 = 19.675$, respectively. The energies obtained are fully converged for both cases (Wille, 1986).

Because the search in the genetic algorithm is nonlinear, we do not need to worry about the degeneracy of the global rotation of the entire system. When we performed the search for the stable configuration of the ionic cluster $(Na^+)_n(Cl^-)_m$ in Chapter 5, we had to remove the motion of the center of mass and the global rotation about an axis through the center of mass, because the search there was in fact a linear search. We can, of course, modify the above method to have one charge sitting at the north pole and another in the xz plane at a variable latitude. The following part of the code shows the assigning of the angles in such a manner.

```
theta[0] = 0;
phi[0] = 0;
theta[1] = Math.PI*r[nv-1];
phi[1] = 0;
int k = 0;
for (int i=2; i<nc; ++i){
  theta[i] = Math.PI*r[k];
  phi[i] = 2*Math.PI*r[k+1];
  k += 2;
}
```

If we replace the corresponding part in the method for the evaluation of the cost, we have removed any global rotation in the simulation.

The Thomson problem is interesting because the number of low-lying excited states increases with the number of charges exponentially. However, we know that the solution is the configuration that spreads out the charges in the most uniform manner; the charges try to avoid each other as much as they can, but confinement on a finite surface forces them to find a compromise. This type of competition is at the heart of the modern theory of the quantum many-body systems. The

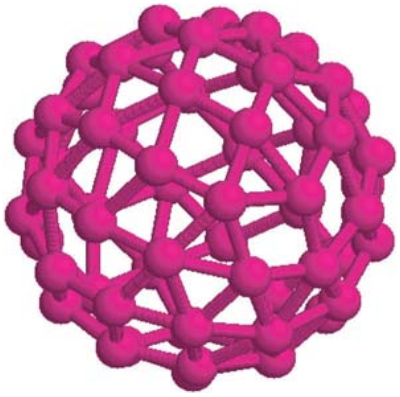

Fig. 11.2 The final configuration of 60 charges on a unit sphere after 60 000 generations of the search under genetic algorithm.

compromise between two conflicting effects drives a system into an exotic state that is typically beyond intuitive thinking or creates surprises. This is perhaps the reason why the Thomson problem is so fascinating. In Fig. 11.2 we show the final configuration of a system of 60 charges, after 60 000 generations. Even though the system is still not fully converged, we can already see the clear distinction between this system and the buckminsterfullerene structure of the molecule C_{60}.

We can, of course, use the same program to search for the stable configuration of even larger systems. The computing time required to obtain a stable configuration will increase, but in principle, we can obtain the stable configuration of multicharges located on the surface of a unit sphere. There are still many interesting issues related to the Thomson problem that are under constant debate, interested readers can find some of these discussions in Pérez-Garrido and Moore (1999) and Morris, Deaven, and Ho (1996).

11.3 Continuous genetic algorithm

The advantage of the binary genetic algorithm lies in its clarity and the transparency of the evolutionary mechanism built in through the three basic operations, selection, crossover, and mutation. However, sometimes the encoding scheme can become unmanageable if accuracy in the array is greatly desired. Examples include cases in which a significant number of local minima are located close together. It then becomes desirable for the variables to remain continuous real parameters and to utilize the machine or language accuracy for the variables.

An alternative to the binary scheme is to code the arrays as real parameters directly. So a chromosome is simply an array of real variables involved in the cost function. The three operations, selection, crossover, and mutation, have to be adjusted in order to work with the real parameters directly instead of operating on the binary strings.

The selection is done in a nearly identical manner because the cost function corresponding to each individual chromosome is always used in any selection scheme. For example, if we still use the tournament method, we can implement the selection using the following method as in the binary scheme.

```
// Method to run tournaments for selecting the parents.
  public static void select() {
    int index[] = new int[ng];
    double d[][] = new double[ng][nv];
    double e[] = new double[ng];
    for (int i=0; i<ng; ++i){
      for (int l=0; l<nv; ++l) d[i][l] = c[i][l];
      e[i] = f[i];
      index[i] = i;
    }
    shuffle(index);
    int k = 0;
    for (int i=0; i<nr; ++i) {
      if (e[index[k]] < e[index[k+1]]){
        for (int l=0; l<nv; ++l) c[i][l]=d[index[k]][l];
        f[i] = e[index[k]];
      }
      else {
        for (int l=0; l<nv; ++l) c[i][l]=d[index[k+1]][l];
        f[i] = e[index[k+1]];
      }
      k += 2;
    }
  }
```

Note that the only difference between the above selection and that of the binary code is that the chromosomes now are stored as real variables instead of binaries.

The crossover can be achieved in several different ways. The simplest is to swap a part of the real arrays. For example, the single-point crossover can be implemented as in the following method.

```
// Method to perform single-point crossover operations.
  public static void cross() {
    Random rnd = new Random();
    int k = 0;
    for (int i=nr; i<nr+nr/2; ++i) {
      int nx = 1 + (int)(nv*rnd.nextDouble());
      for (int l=0; l<nx; ++l){
        c[i][l] = c[k][l];
        c[i+nr/2][l] = c[k+1][l];
      }
      for (int l=nx; l<nv; ++l){
        c[i][l] = c[k+1][l];
        c[i+nr/2][l] = c[k][l];
      }
      k += 2;
    }
  }
```

Note again that the code appears to be virtually identical to that of the binary code, but the crossover is not happening at an arbitrary point that can be the middle of a binary string. Instead, it is at a selected location of the real array that represents a chromosome. The crossover in the binary code allows the change to happen in the middle of a binary segment that represents a single real variable. But the real-parameter version above swaps the variables themselves. There are other ways to implement crossover that allow a partial change of a real variable (Goldberg, 1989).

The most significant departure from the binary scheme is in the mutation operation. When we have the binary representation of a chromosome, binary bits are natural and we expect a bit to change from true to false or vice versa if mutation occurs. But when the chromosomes are given in real values, it is not entirely clear what change a variable will be subject to if mutation occurs. One way to implement mutation is with a random number. For example, if a real number is chosen to mutate according to the percentage of mutation specified, we can replace it with a uniform random number to the region [0, 1]. Note that all the variables are restricted to the region [0, 1]. The replacement with a random number would be equivalent to having a percentage of bits altered if the variable were represented by a segment of binaries. The following method is an implementation of such a mutation scheme.

```
// Method to mutate a percentage of bits in the selected
// chromosomes except the best one.

  public static void mutate() {
    Random rnd = new Random();
    double r[] = new double[nv];

 // Mutation in the elite configurations
    for (int i=0; i<ne; ++i) {
      for (int l=0; l<nv; ++l) r[l] = c[i][l];
      int mb = (int)(nv*pm+1);
      for (int j=0; j<mb; ++j){
        int ib = (int)(nv*rnd.nextDouble());
        r[ib] = rnd.nextDouble();
      }
      double e = cost(r);
      if (e<f[i]){
        for (int l=0; l<nv; ++l) c[i][l] = r[l];
        f[i] = e;
      }
    }

 // Mutation in other configurations
    int mmax = (int)((ng-ne)*nv*pm+1);
    for (int i=0; i<mmax; ++i) {
      int ig = (int)((ng-ne)*rnd.nextDouble()+ne);
      int ib = (int)(nv*rnd.nextDouble());
      c[ig][ib] = rnd.nextDouble();
    }

 // Rank the chromosomes in the population
    rank();
  }
```

When the above changes were implemented in a real-parameter genetic algorithm and applied to the Thomson problem, it was found that the scheme was as powerful as the binary version. The advantage of the real-parameter code is that it avoids having to encode and decode the chromosomes back and forth every time the variables are used or stored and therefore saves a large portion of computing time.

11.4 Other applications

The potential of the genetic algorithm was not realized immediately when the scheme was first introduced (Holland, 1975), but it picked up the pace afterwards (Goldberg, 1989). Now the scheme is applied in many fields, including business, economics, social studies, engineering, biology, physical sciences, computer science, and mathematics. In order to show its importance, we highlight a few applications in this section. The discussion is far from being complete; interested readers should search the current literature to obtain a better glimpse of what is going on with the genetic algorithm.

Molecules, clusters, and solids

The Thomson problem can be generalized in the search for stable structures of other small clusters of atoms and molecules that can be well described by a classical n-body interaction potential $V(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_n)$, which is usually obtained from some types of first-principles calculations, such as the glue potential for gold clusters (Ercolessi, Tosatti, and Parrinello, 1986).

The simplest n-body interaction is an isotropic pairwise (two-body) interaction, such as the Coulomb interaction involved in the Thomson problem. For example, the interaction between two inert gas atoms is well described by the (two-body) Lennard–Jones potential

$$V(r_{ij}) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right], \tag{11.6}$$

which we introduced in Chapter 8 to illustrate the molecular dynamics simulation. We can follow what we did with the Thomson problem to find the stable geometric configurations of clusters of inert gas atoms. However, there is one change that must be made in order to accommodate these unrestricted clusters. The particles are allowed to move along the r direction as well as the θ and ϕ directions. We can still keep all the variables finite by introducing a cut-off radius r_0, which can be viewed as the largest distance the particle can reach from the center of the clusters. Unless the cluster is falling apart, this choice of cut-off radius does not affect the stable configuration of the clusters. The following method shows how the cost is calculated in the program.

Fig. 11.3 The stable structures of five and eight particles that interact with each other through the Lennard–Jones potential.

```
// Method to evaluate the cost for a given variable array.

  public static double cost(double r[]) {
    double g = 0;
    double ri[] = new double[nc];
    double theta[] = new double[nc];
    double phi[] = new double[nc];

    int k = 0;
    for (int i=0; i<nc; ++i){
      ri[i] = r0*r[k];
      theta[i] = Math.PI*r[k+1];
      phi[i] = 2*Math.PI*r[k+2];
      k += 3;
    }

    for (int i=0; i<nc-1; ++i){
      double rhoi = ri[i]*Math.sin(theta[i]);
      double xi = rhoi*Math.cos(phi[i]);
      double yi = rhoi*Math.sin(phi[i]);
      double zi = ri[i]*Math.cos(theta[i]);
      for (int j=i+1; j<nc; ++j){
        double rhoj = ri[j]*Math.sin(theta[j]);
        double dx = xi - rhoj*Math.cos(phi[j]);
        double dy = yi - rhoj*Math.sin(phi[j]);
        double dz = zi - ri[j]*Math.cos(theta[j]);
        double r6 = Math.pow((dx*dx+dy*dy+dz*dz),3);
        double r12 = r6*r6;
        g += 1/r12-2/r6;
      }
    }
    return g;
  }
```

Note that the variables are still kept in the region of $[0, 1]$ for convenience. Because of this choice, the main methods for selection, crossover, and mutation are unchanged. In Fig. 11.3, we show the stable configurations of the Lennard–Jones clusters with five and eight particles. We have used the real-parameter version of the genetic algorithm discussed in the preceding section in the search, and ε and $2^{1/6}\sigma$ as the units of energy and length, respectively. The potential energies per particle are $E_5/5 = -1.8208$ and $E_8/8 = -2.4776$. Note that even though the clusters studied here are extremely small, they already appear to be

a stack of tetrahedrons that are the building blocks of a face-centered cubic or hexagonal close-packed lattice.

The search can be enhanced with a better design of the crossover operation, especially for large clusters. One type of such is called basin hopping, in which the potential energy surface has many local minima and the new configurations are created to overcome the barriers between these basins by swapping certain particles from different configurations. For example, we can cut two different configurations of the cluster through a randomly chosen plane that goes through or is near the center of mass of the cluster, splitting the cluster into two equal halves of compensated parts, and then exchange the parts to create two offspring (Deaven and Ho, 1995). This catchment basin transformation of the potential energy surface can also be built into other optimization schemes (Wales and Scheraga, 1999).

We can also use the genetic algorithm to search for the stable lattice structure of a solid. There is an additional issue that needs to be addressed. In practice, we cannot deal with an infinite system directly. However, we can always put the particles in a box and then impose the periodic boundary condition on the system under study. Because we are searching for a crystal structure, we also need to find ways to relax the size and the shape of the simulation box, as we did in Chapter 8 for the molecular dynamics simulation of an infinite system. In fact, we can combine the genetic algorithm with a typical simulation technique, such as molecular dynamics, or an ab initio total energy calculation from, for example, density function theory in the study of the phase transition of bulk materials. For example, a combination of the genetic algorithm and a solution of the Ornstein–Zernike equation provides an effective exploration of the phase diagram of spherical polyelectrolyte microgels (Gottwald *et al*., 2004), and a combination of the genetic algorithm and the density functional theory allows an extensive search for the most stable four component alloys out of 192 016 possible fcc and bcc structures formed in 32 different metals (Jóhannesson *et al.*, 2002).

Chaos forecast

There have been many applications of genetic algorithm in all sorts of nonlinear processes, including in the study of dynamical systems. For example, if there is a given time sequence $(y_1, y_2, \ldots, y_n)$ that can be either a data set from an experimental measurement or one from a computer simulation, we can use the genetic algorithm to obtain the information hidden in the sequence. A successful analysis of the sequence would provide a good description of the nonlinear terms involved and the nature of the sequence, linear, periodic, random, or chaotic.

The fundamental problem here is to find the appropriate function form of the time dependence in the dynamic variable

$$y_i = f(y_{i-1}, y_{i-2}, \ldots, y_{i-k}) \tag{11.7}$$

from the given time sequence. Here k is an assumed number of previous steps that determine the current dynamics. The idea is to decompose $f(y_{i-1}, y_{i-2}, \ldots, y_{i-k})$ into all possible building blocks and let a selection process that follows the genetic concept take place. After generations of evolution, we expect good blocks to be amplified and bad blocks eliminated (Szpiro, 1997; López, Álvarez, and Hernández-García, 2000). As soon as we find the exact form of the function, or the closest one possible, we can forecast the future behavior of the dynamical system, including chaos.

Depending on how much information in $f(y_{i-1}, y_{i-2} \ldots, y_{i-k})$ is known, we can construct an approximate cost for the search process. If we already know the model, namely, the function form of $f(y_{i-1}, y_{i-2} \ldots, y_{i-k})$, we can find the precise values of the parameters involved in the model by constructing a chromosome that represents a specific choice of the parameter set. For example, if we have l parameters, $v_1, v_2, \ldots, v_l$, in the model function $f(y_{i-1}, y_{i-2}, \ldots, y_{i-k})$, the cost can be chosen as

$$g(v_1, v_2, \ldots, v_l) = \sum_{i=k+1}^{n} (y_i' - y_i)^2, \tag{11.8}$$

where y_i' is the predicted value based on the given set of parameters in the whole sequence and y_i is the actual value. Then we can follow the three operations, selection, crossover, and mutation, in the genetic algorithm to tune the parameter set in order to find the appropriate ones.

When we do not know the exact form of $f(y_{i-1}, y_{i-2}, \ldots, y_{i-k})$, we can create chromosomes from the results of certain mathematical operations on the variables $y_{i-1}, y_{i-2}, \ldots, y_{i-k}$. Following the evolution of the chromosomes, we can at least find an approximate form of $f(y_{i-1}, y_{i-2}, \ldots, y_{i-k})$ that can describe certain aspects of the system, even if it is chaotic (Szpiro, 1997). The mathematical operations can be the four basic operations, addition, subtraction, multiplication, and division, or more advanced operations, such as logarithmic, trigonometric, or logic operations. Care must be taken to avoid forbidden operations, such as division by a zero or taking the square root of a negative quantity in the case of real operations only. The selection can be achieved in the manner we have discussed, but the crossover can only be done by swapping certain building blocks or operations. This is quite similar to the real-parameter version of the genetic algorithm scheme. Mutation is achieved in a closely related manner by randomly swapping a given percentage of two operations or two building blocks that are also randomly selected.

Best strategy in a game

It is commonplace for a corporation to apply the concept of game theory in developing its business strategies. Game theory is the study of all the possible responses that each individual can make in a game (a process that involves

mutually interacting rational players) and the consequence (gain or loss) of each response (Osborne, 2004). The essential part of a successful analysis of a game is figuring out the strategy for an individual player to obtain the maximum gain in the long run, based on the available record.

Here we use a simple example of the adaptive minority game (Sysi-Aho, Chakraborti, and Kaski, 2004) in order to illustrate the potential of applying the concepts of the genetic algorithm. The simple minority game has an odd number of N players with only two possible actions that can be recorded as a binary 0 or 1. A player wins a point by ending up in the minority group. Assuming that all the players have access to the M most recent results with each recorded also as a binary, 0 for the winning of 0's group and 1 for the winning of 1's group. So the record is one of the 2^M possible combinations. For each combination, a player can choose either 0 or 1, that makes a total of 2^{2^M} possible strategies for each player. However, each player is only allowed to have a finite number of S strategies in his strategy pool from which to choose. The adaptive minority game allows a player to modify his strategy pool. This modification can be constructed according to the genetic algorithm. For example, we can map each strategy into a gene that contains a binary string of a possible record and action. The chromosome is a binary string of the genes of all the agents at a given time t, which is increased by a unit each time the game is played. Among different strategies, we can create a population at a given time. The performance of the system is measured by the time-dependent cost (called the utility)

$$U(x) = \frac{1}{x_0}\left[x + \Theta(x - 2x_0)(N - x)\right], \tag{11.9}$$

where $x \in [1, N]$ is the number of agents that take a specific action (1 or 0) at time t, $x_0 = (N-1)/2$ is the maximum number of possible winners, and $\Theta(y) = 1$ for $y > 0$, zero otherwise. The maximum cost is normalized to 1 and decreases if x deviates from the average $N/2$. It has been shown that a single-point crossover operation can improve the performance of an individual agent as well as the system as a whole (Sysi-Aho, Chakraborti, and Kaski, 2004). It would be interesting to see whether the selection and mutation operations are able to do the same.

The minority game is not unique in the sense of having the characteristic that there is a gain for each individual and an overall efficiency of the system. We can identify many economical and social phenomena that carry such a generic feature. More applications of the concepts of genetic algorithm are expected to emerge in near future.

11.5 Genetic programming

The problems involved in game theory and chaos forecasting are more complicated than that of finding the stable configuration of an atomic cluster; they

require optimization of the processes involved. If we imagine that each of the strategies taken by an agent in the minority game is a possible computer operation (represented there by a binary string), the entire sequence of strategies can be viewed as a piece of symbolic computer program and the sequence that receives the best overall performance is the best available computer program to achieve the task of the minority game.

The scheme that selects the best set of operations (best program), based on the concept of evolution, from all the possible sets of operations (possible programs) for solving a certain problem or achieving a certain task is called genetic (or evolutionary) programming, which was first introduced in the early 1960s (Fogel, 1962; Fogel, Owens, and Walsh, 1966). The scheme has experienced much renovation since then (Koza, 1992; 1994). For an introduction to the subject, see Jacob (2001). The main difference between genetic programming and the genetic algorithm is that the genetic algorithm tries to find the configuration (chromosome) that optimizes the cost, but genetic programming tries to find the best process among all the possible processes according to the assigned fitness. So genetic programming creates computer codes for certain objectives and modifies them according to evolutionary principles. This is much more proactive in the sense of creating a certain artificial intelligence in the programs.

For example, if we want to design software that can interpret the Chinese characters written with an electronic pen on a pat that is formed with a matrix of capacitors and resistors, the cost (the measure of fitness) function could be the percentage of characters that have been interpreted wrongly since the beginning of the writing. This is a difficult task because the software must be able to recognise the characters even when there are differences in the hand-writing. But this is the type of the problem for which genetic programming works the best, with a simple fitness function that depends on a large number of variables from different handwriting styles.

Genetic programming involves roughly the same pieces of ingredients as in the genetic algorithm. First, an initial pool of computer programs, formed with *functions* (computational operations) and *terminals* (elements to be operated on) of the possible solutions of the problem, is created. Parents are then selected according to the fitness of each program. Finally, new generations are created with crossover and mutation. The key difference between the genetic algorithm and genetic programming is what is being modified – configurations of the variables in a genetic algorithm but computational operations or elements in a genetic program.

A typical function in genetic programming is an arithmetic operation, such as addition, division, or taking a square root, but it can also be a more sophisticated operation, like a function operation $z = f(x, y)$ on the two terminals (elements) x and y. The simplest terminal is typically a variable or parameter; a general terminal is a combination of many variables and parameters through functions. For example, if the function is $z = e^x \ln(y - 1)$, we can consider the terminals to be x and y, which can be combinations of some other variables and parameters,

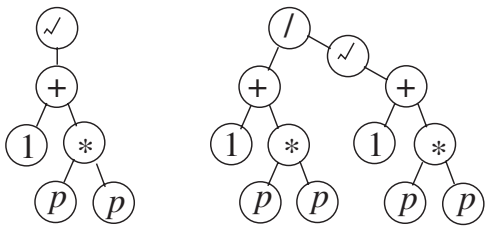

Fig. 11.4 Two different tree diagrams that represent the same energy equation.

such as $x = a^2 + 2b - \sqrt{c}$ and $y = 3a + b^2 - \ln c$. Of course, $y > 1$ and $c > 0$ if we are dealing with real quantities only.

One of the difficulties in designing a genetic program is in figuring out an appropriate fitness (cost) associated with each possible program. The cost can only be evaluated through running the program. In some cases in order to obtain a fair estimate of the cost, a program must be run multiple times, resulting in a time consuming process.

Let us take an extremely simple example in physics to highlight how genetic programming works. Consider that we are searching for the relation between the energy E and momentum p of a relativistic particle of mass m, which is given as

$$E = c\sqrt{m^2c^2 + p^2}, \tag{11.10}$$

where c is the speed of light in vacuum. If we use mc^2 as the unit of energy and mc as the unit of momentum, we have $E = \sqrt{1 + p^2}$. We can formally use a *tree diagram* to represent the actual equation. Note that each function in this problem can only have one or two terminals. When there are two terminals, the operation of the function is binary, from left to right. There are many possible representations (tree diagrams) for the equation, two of which are shown in Fig. 11.4.

For simplicity, we assume that there is a cut-off for the momentum $p \leq p_c$ and the functions are five elementary operations: addition $(+)$, subtraction $(-)$, multiplication $(*)$, division $(/)$, and taking the square root $(\surd)$. We further assume that the equation is given at n discrete points with $E_i = E(p_i)$, for $i = 1, 2, \ldots, n$. Now we can create the initial pool of equations by treating p and 1 as the only variable and parameter that can be used repeatedly. We can limit the number of the uninterrupted segments in the longest branch in the tree diagram to four. Note that we also must avoid dividing by a zero or having a negative value within a square root. We can simply eliminate that diagram if it is the case.

After we have the initial pool, we can assign a fitness (cost) to each of the diagrams based on

$$f = \sum_{i=1}^{n} \Delta E_i^2, \tag{11.11}$$

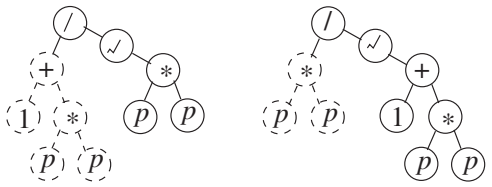

Fig. 11.5 Two possible parents that can be used to create offspring by swapping the dashed branches.

where ΔE_i is the energy difference between a tree diagram in the pool and that of actual value. Based on the cost, we can run tournaments to select parents.

Assume that we have found two parents as given in Fig. 11.5. We can perform crossover to create offspring. The crossover is performed between the two parents at similar functions, two-terminal or one-terminal. For example, if the crossover is done by swapping the dashed branches, we obtain two offspring, with one of them being the exact equation sought. Note that a crossover can also be performed on two identical parents with different parts swapped, which is equivalent to swapping different segments of two identical chromosomes in a genetic algorithm.

Mutation can be performed at each vertex. For example, if a function is selected randomly to be mutated, we can use one of the five operations to replace it. If the variable is selected to be mutated, we can replace it with the parameter, or vice versa. Note that we still have to keep the equation valid, that is avoid having a negative value inside a square root or dividing a quantity by a zero.

There have been much activity in genetic programming, including progress in different methods of assigning cost, performing crossover, and carrying out mutation. The field is still fast growing, especially in the areas of artificial intelligence and machine learning. Interested readers can find detailed discussions on these subjects in Ryan (2000), Langdon and Poli (2002), and Koza *et al*. (1999; 2003).

Exercises

11.1 Assuming that the interaction between Na^+ and Cl^- in a NaCl molecule is given by

$$V(r) = -\frac{e^2}{4\pi\epsilon_0 r} + V_0 e^{-r/r_0},$$

with $V_0 = 1.09 \times 10^3$ eV and $r_0 = 0.330$ Å, find the bond length of the molecule with the genetic algorithm.

11.2 Find the minimum of $f(x, y, z) = y \sin(4\pi x) + 2x \cos(8\pi y)$ in the region of $x, y \in [-1, 1]$ with the genetic algorithm.

11.3 Find the stable geometric structures of clusters of ions $(Na^+)_n(Cl^-)_m$ with small integers n and m by a genetic algorithm search.

11.4 Search for the stable geometric structures of small Lennard–Jones clusters. Specifically, find the Mackay icosahedron for 55 particles and decahedron for 75 particles. Are the Mackay icosahedron and decahedron the stable structures of the given systems?

11.5 Using the four basic operations, addition, subtraction, multiplication, and division, and the data points from the previous time steps, find the correct expression for the right-hand side of the Duffing model

$$\frac{d^2x}{dt^2} = b\cos t - g\frac{dx}{dt} - x^3$$

and forecast chaos.

11.6 Create a long set of data points with an equal-time step for a simple harmonic oscillator, $(\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_n)$, where $\mathbf{y}_k$ for $k = 1, 2, \ldots, n$ are two-component vectors, with one component for the angle of the pendulum away from the vertical and the other for its time derivative. Determine the optimal function form of $\mathbf{f}$ from

$$\mathbf{y}_i = \mathbf{f}(\mathbf{y}_{i-1}, \mathbf{y}_{i-2}, \ldots, \mathbf{y}_{i-k}),$$

for $k = 1, 2, 3, 4$. What happens if the pendulum is a driven pendulum with damping?

11.7 Generate a long, random set of data and determine that it cannot come from any dynamical system. A specific model can be used as a test.

11.8 Develop a genetic algorithm program to play the optimal minority game. When each of the three operations, selection, crossover, and mutation are analyzed, which one is most critical?

11.9 Use genetic programming to create a solution of a two-dimensional maze built from $n \times n$ squares.

11.10 The Chinese board game *Weiqi* has 361 vertices through 19×19 perpendicular lines. Two players take turns to occupy the vertices, one at a time with two types of stones (black and white), one for each player. If one player completely surrounds one or more of his opponents stones, he removes the surrounded stones. The winner is the one who occupies most vertices at the end. Using the genetic algorithm or genetic programming, develop a computer program that can play Weiqi against a skillful human player.